

Semantics (I)

Practical Expression Semantics

- Just getting the syntax (structure) of the language right doesn't mean you are generating code that works correctly. (run-time errors or unexpected results)
- You need to understand not just how to write a program, but what each part of the program actually does.

```
// funnyIncrement.c
//

#include <stdio.h>

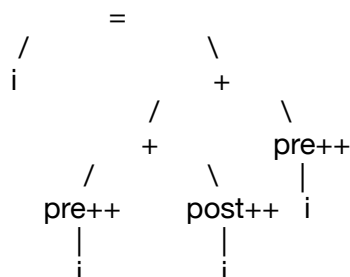
void foo () {
    int i = 4;
    i = ++i + i++ + ++i;
    printf("foo: %d\n", i);
}

int main () {
    foo();
    return 0;
}
```

- Example:
 - Here is a C program that declares an int with value 4, and increment it and assignment it to itself. (show funnyIncrement.c)
 - What are the possible outputs of the program? [17 on my mac, 18 on gloin]



- The confusion here is when to execute the pre-increment operators.
- If we draw a simplified concrete syntax tree, it will be a tree like this



- my mac: execute left to right when the node is reached during tree traversal
- gloin: execute all pre- and post-increment before traversal

- The tree can handle operator precedence. However, it does not guarantee the order or timing of the execution of the sub-trees.
- Executing the sub-trees in different orders can produce different results.
- C language specification **does not define** the behavior of the program where there is **more than one pre- or post- increment operator applied to a single variable within a single expression**.
- The behaviors end up being **compiler specific**. The semantic meaning of the program is undefined.
- How about Java?
 - Let `int x = 3; x += x++ + ++x;` This is legal in Java, so the syntax is correct.
 - What about the semantics? After the two statements have been executed, what is the value of `x`?
 - The value of `x` is **precisely defined in Java**, which is 11. `x = 3 + 3 + 5`.
 - **Increment and decrement** operators have a **higher precedence than** the other **mathematical operators**. They are applied from **left to right in the order in which they occur** unless one of them has been given a higher precedence through the parentheses.
- Example:
 - `if (x++ == x++) {System.out.println("Yes");}`
 - This condition is always false in Java.
 - `if (++x == x++) {System.out.println("Yes");}`
 - This condition is always true in Java.
- **Lazy evaluation**
 - What is the output of the following C code? Will the function `foo` be invoked?
 - The `foo` function will not be invoked, and the output is "Expecting the second ..."

```
// LazyEval.c
#include <stdio.h>

int foo () {
    printf("Hi there!\n");
    return 1;
}

int main () {
    if (0 && foo()) {
        printf("What's up!\n");
    }

    if (1 || foo()) {
        printf("Expecting the second spring break!\n");
    }

    return 0;
}
```

- Why the foo function wasn't invoked? [Lazy evaluation]
- Most languages now use **lazy evaluation** (short circuit evaluation). When evaluating boolean expressions, **do it from left to right and stop as soon as you know the value of the expression**. Therefore, in the example above, we don't need to call foo(). [C, Java, Python uses lazy evaluation]
- Example:
 - Given the following Java and Python code, which program will generate an error? [Java will generate a DivideByZero error]

```
//LazyEval.java
public class LazyEval {
    public static void main (String[]
args) {
        int x = 0;
        int y = 4;
        if (y/x > 3 || x != 0) {
            System.out.println("Hi,
there!");
        }
    }
}
```

```
# LazyEval.py
x = 0
y = 4
if (x != 0 and y/x > 3):
    print("Hi there!")
```

- Both programs divide y by zero, why only the Java code generate error?
 - Because of the lazy evaluation, the second half of the expression in the python code didn't execute.
- Therefore, **the order of expressions in a lazy evaluation matters**.
- Why do most PLs support Lazy evaluation? [make the program more efficient]

What is semantics? How do we define semantics?

- The **semantics** of a programming language is a **precise definition of the meaning** of any program that is **syntactically and type-wise correct**
- There are at least three ways to define the semantics
 - **Operational semantics**: the **meaning** of a program **is the output** produced by a given architecture/compiler pair (pre- and post- increment in C)
 - **Axiomatize semantics**: the **meaning** of a program **can be** rigorous **proven** by using a systematic logical argument (formal specification)
 - **Denotational semantics**: the **meaning** of a program can be described as a **collection of meaning functions operating on the program state** (this course)

What is the program state?

- The **state** of a program is the **binding of all variables to their current values**.
- A variable and its value can be modeled as an **ordered pair**, e.g. **<i, 5>**
- A state is a **set of ordered pairs** (dictionary)

$$state = \{ \langle var1, val1 \rangle, \langle var2, val2 \rangle, \dots, \langle var_m, val_m \rangle \}$$

- A program can have more than one states.
 - A state is a snapshot of the memory of the variables used by a program. After each statement execution, the program will have a new snapshot.
- Example
 - The program calculates factorial. Show the states of this program.

```

1 int main () {
2   int a, b;
3
4   a = 1;
5   b = 2;
6
7   b = a + b;
8
9   return 0;
10 }
```

Trace Table

Snapshot	Before Statement	State
1	4	{<a, undef>, <b, undef>}
2	5	{<a, 1>, <b, undef>}
3	7	{<a, 1>, <b, 2>}
4	9	{<a, 1>, <b, 3>}